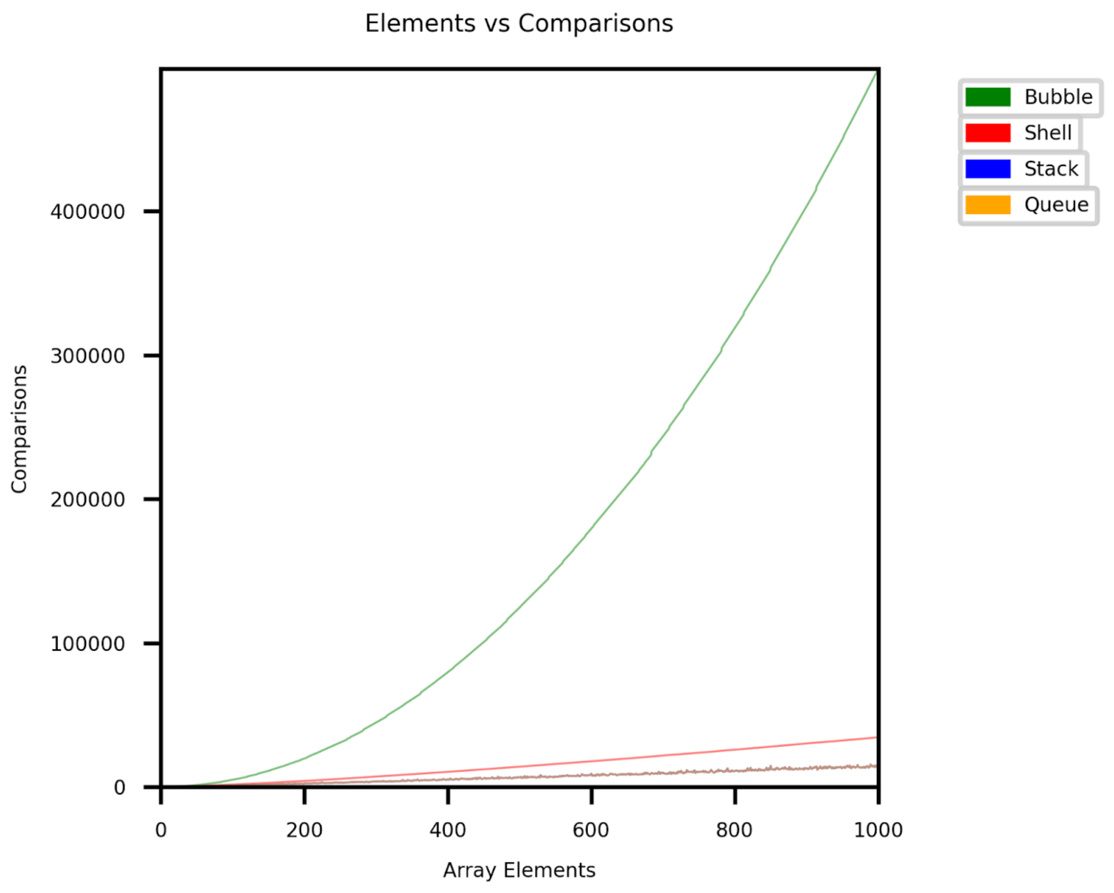
Nicholas Chan
nipchan@ucsc.edu
4/25/2021
CSE13s Spring '21

Writeup:
Assignment 3: Sorting

For assignment 3, I implemented and compared the performances of a bubble sort, shell sort, quicksort with a stack and a quicksort with a queue using metrics such as the counts of moves and comparisons made. The first thing I did was test out each of these sorting algorithms using the python pseudo code provided to us in the asgn3 instructions pdf. After successfully implementing them in python, the implementations for bubble sort and shell sort were fairly straightforward to get done in c. However before moving onto the quicksort algorithms, I spent some time figuring out how the stack and queue abstract data types were created, stored, used and deleted. Once I made a working stack and queue (I tested my stack and queue using my stack/queue print functions which can be found in my stack.h and queue.h header files) I was able to implement them in c.

The graphs below display the statistics I received from testing each previously mentioned sorting algorithm for 1000 iterations. Each iteration would task an algorithm with sorting a random integer array of a size determined by the number of the iteration that the algorithm was testing.

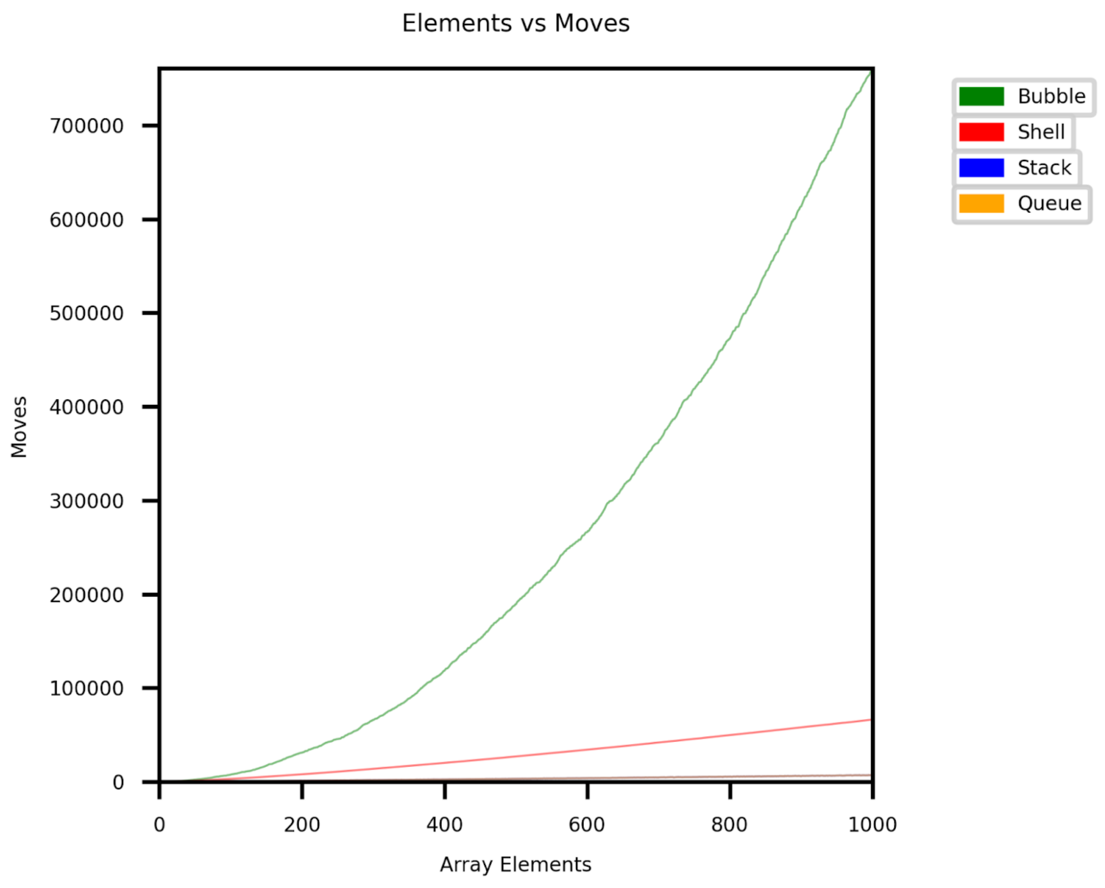


Graph comparing the comparisons made by the bubble, shell, quicksort (stack) and quicksort (queue) sorting algorithms over array sizes ranging from 0 to 1000.

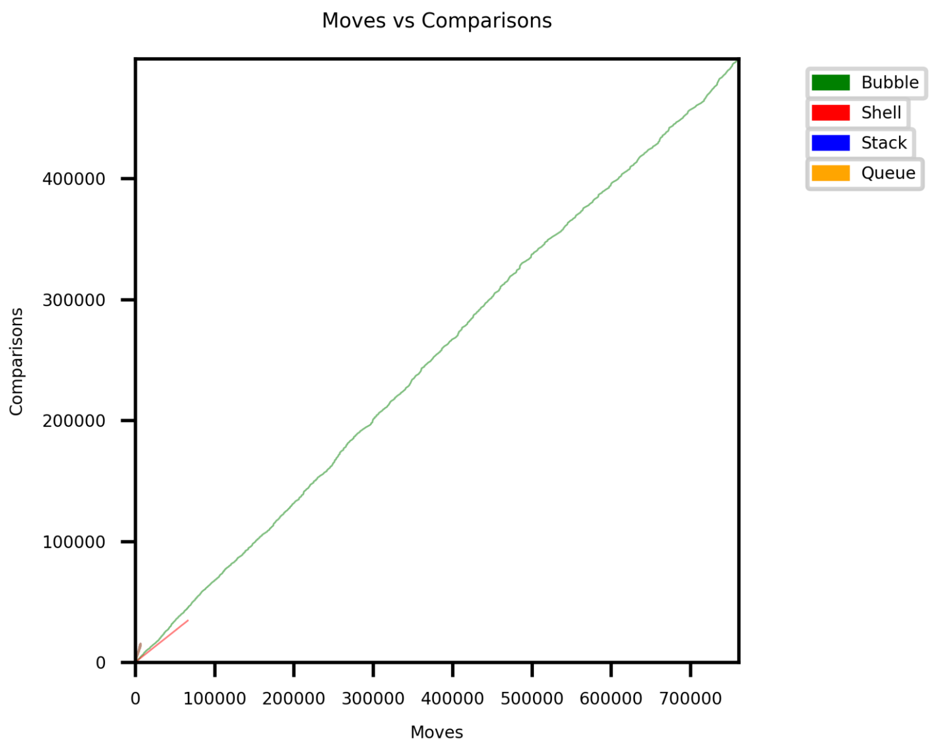
By looking at how the bubble sort algorithm was put together from the python pseudocode I was given and after implementing it myself in C, I speculated that it would have had the highest time complexity in comparison to the rest of the algorithms I saw in assignment 3. This speculation was also strengthened by many of the lessons on time complexity in programs and bubble sort's naive approach towards sorting an array. My reasoning for this high time complexity was based on the nature of bubble sort's array item comparison process. Bubble sort's nested for loop had no exit condition except for exhausting every item it would have had to iterate over to reach a comparison. This condition makes it easy for bubble sort to reach a high time complexity when sorting a scrambled/randomly shuffled array of integers. The graph of the comparisons made by each sorting algorithm confirms my initial speculation as well as the many claims made in lecture. The trend that bubble sort's statistics seem to take resembles that of a program of $O(N^2)$ time complexity.

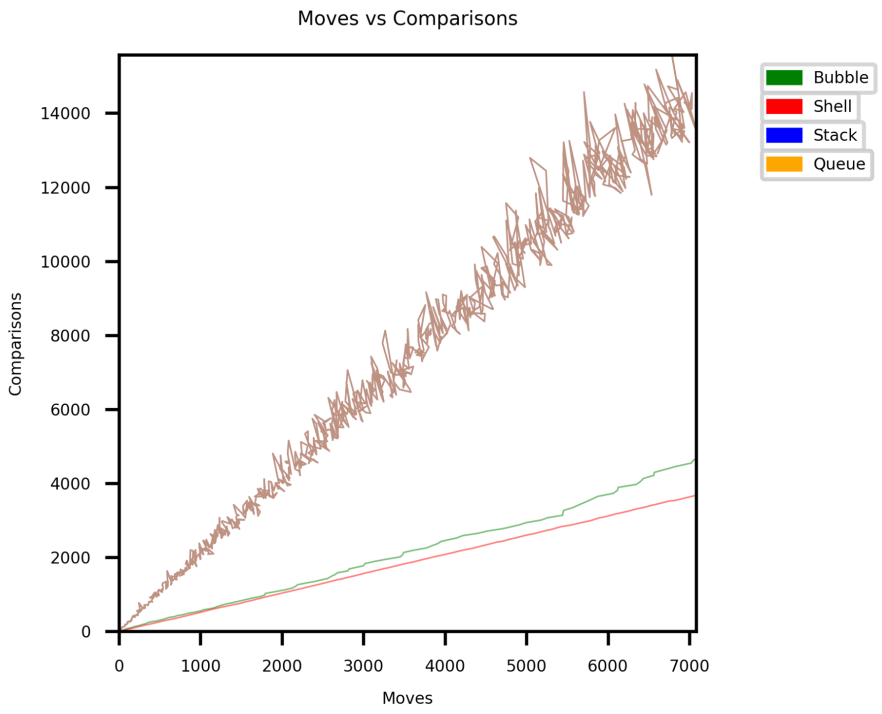
Shell sort is the next sorting algorithm I learned about in this assignment. It contains a while loop nested inside a for loop which is nested inside a for loop (dependent on the gap sequence you use) which may give off the impression that it would have a pretty high time complexity. However there is an exit condition that becomes closer to reach with each iteration of the while loop. Also to note is that the outermost for loop is limited by the size of the array you task shell sort with sorting. These conditions for the shell sorting algorithm combine to create the action of comparing array items over a gap width which is changed on each passing of the outermost for loop (from the pseudocode provided in the asgn3 instructions pdf). This mechanism of shell sort prevents the need for an excessive number of comparisons by making more meaningful moves out of each comparison. The trend that shell sort's statistics seem to take resembles those of a program around O(nlogn) time complexity, maybe a little worse because of the mandatory for loop over gaps in the gap array at the beginning of the algorithm. Because of this and the first for loop which iterates over a gap range, it would seem that shell sort would be more time complex than a O(nlogn) program.

The quick sort algorithm was the last I implemented in this assignment, since the implementations of quicksort with a stack and queue both performed identically in terms of moves and comparisons, I will bring up how they are different later. Quicksort utilizes a partitioning sub-routine to conduct comparisons, moves and swaps. Quicksort's structure consists of a while loop over the length of a stack that is updated by the partition subroutine. The partition subroutine consists of a while loop comparing an upper and lower bounds. Inside that first while loop are two parallel while loops nested inside and are checked by a comparison based on pivot positions and array items. The combination of the subroutine within the quicksort function gives me the impression that it takes an O(nlog(n)) time complexity. Also, since merge sort is a "divide and conquer" algorithm like Merge sort, which was brought up in lecture, it seems likely that O(nlog(n)) is the time complexity of quicksort. This is supported by the structure of the algorithm and its position in the graphs above relative to other algorithms.

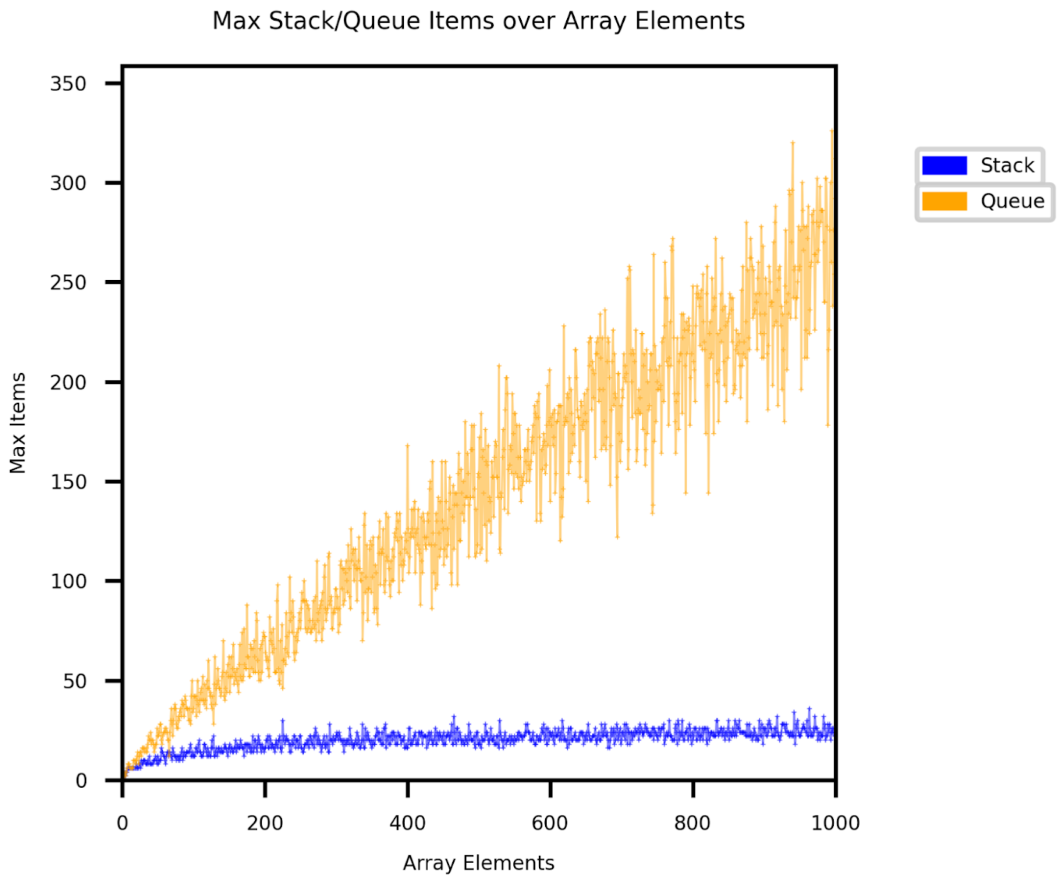


Movements of array items are dependent on comparisons being made, but depending on the type of algorithm, the ration between movements and comparison will vary. This notion becomes apparent by taking a look at graphs of algorithm moves over array element sizes. For visual proof of this difference in the ratios of moves to comparison exhibited by each of the sorting algorithms, see the images below. From the trends of the curves of these images, it appears that each algorithm follows its own ratio. The first image below is the full scope with the most moves and comparisons taken into account while the second image is the fitted scope with the minimum amount of moves and comparisons taken into account.





From these graphs, the ratios of comparisons to moves seem to go from quicksort (stack), quicksort (queue), Bubble sort and shell sort in descending order. Ratios of moves to comparisons would be vice versa. These graphs can help interpret the meaningfulness of each comparison or each move based on how many comparisons were needed per sort and how many comparisons or sorts were needed to complete a sorting task. For example since shell sort's trend was lower overall than bubble sort's, we could infer that shell sort needed less comparisons to make a move than bubble sort did on average. Looking at the full scale graph, we will also see that shell sort finished with much less moves and comparisons than bubble sort.



Although the performances of quicksort (stack) and quicksort (queue) were identical when looking at elements per move and comparisons per move, here we can see that they differed in the maximum amount of items that they were able to store in their respective ADT's (either stack or queue). It appeared that as array sizes increased, the maximum number of items in an ADT at a time increased more with queues than with stacks. Stacks seemed to max out at around 25-30 array items while queues seemed to max out at around 250-300 array items. A difference to notice about stacks and queues is that queues are accessed through two indices (head and tail) while stacks only have one index (the top). This difference is likely the root of the discrepancy between the maximum number of items held in queues vs stacks as the rest of the operations applied to stacks and queues are virtually identical. Also in the algorithm for quicksort, the implementations for stacks and queues differ in the order that "hi" and "lo" are removed in order to account for the different systems for adding and removing array items.

Overall, it seems that quicksort (both the stack and queue implementations) live up to their namesake when compared to bubble sort and shell sort. I believe that quicksort (stack), quicksort (queue), shell sort and bubble sort is the correct order of the algorithms, going from the least time complexity to the highest time complexity. This is supported by the graphs I was able to produce and the structures underlying each of the sorting algorithms.